UCRL-92325
Preprint

A Butterfly Processor-Memory Interconnection for a Vector Processing Environment

Eugene D. Brooks III

This paper was published in
Parallel Computing
Volume 4, pages 103-110
1987

February 1985

A Butterfly Processor-Memory Interconnection for a Vector Processing Environment[†]

Eugene D. Brooks III

Parallel Processing Project
Lawrence Livermore National Laboratory
Livermore, California 94550

ABSTRACT

A fundamental hurdle impeding the development of large N common memory multiprocessors is the performance limitation incurred in the switch connecting the processors to the memory modules. Multistage networks currently considered for this connection have a memory latency which grows like $\alpha \log_2 N$[*]. For scientific computing, it is natural to look for a multiprocessor architecture that will enable the use of vector operations to mask memory latency. The problem to be overcome here is the chaotic behavior introduced by conflicts occurring in the switch. In this paper we examine the performance of the butterfly or indirect binary n-cube network in a vector processing environment. We describe a simple modification of the standard 2x2 switch node used in such networks. This local modification to the switch node endows the network with a surprising global property. It adaptively removes chaotic behavior during a vector operation.

February 1985

[†]Work performed under the auspices of the U. S. Department of Energy by the Lawrence Livermore National Laboratory under contract No. W-7405-ENG-48.

[*]By using n*n switch nodes [5] memory latency can be reduced to $\alpha \log_n N$

A Butterfly Processor-Memory Interconnection for a Vector Processing Environment[†]

Eugene D. Brooks III

Parallel Processing Project
Lawrence Livermore National Laboratory
Livermore, California 94550

1. Introduction

The VLSI revolution, which has drastically reduced the cost of computer circuits, is making large N multiprocessor designs possible. VLSI implementations of pipelined processors have become so cheap that cost is no longer the factor inhibiting the development of high performance shared memory multiprocessors. One of the stumbling blocks preventing the development of such machines is the problem of keeping a large number of pipelined processors adequately fed from a shared memory. This is the topic which we will deal with in this paper. Unfortunately, other problems exist. Synchronization cost, discussed in [1], must also be dealt with before such machines will become a reality.

There are several techniques which can be used to connect processors to the memory modules of a multiprocessor. We consider how to make effective use of the butterfly or indirect binary n-cube packet switching network, shown in figure 1, to do this. Other authors [3-5] have investigated the properties of δ and Banyan networks of which the butterfly is a member. Of particular note is the work of Dias and Jump, where the value of adding buffers to the switch nodes of the network was clearly demonstrated. We take a systems approach in our investigation. The packet switching network is just one part of the parallel computer architecture which must be considered as a whole. Only by considering the system of interacting components, in this case the user application, the cpu nodes, the switching network and the memory modules, can one put the performance analysis of the switching network into context.

As one of the fundamental problems imposed by the switching network is latency, we consider the use of vector operations in multiprocessors. The use of vectorization to deal with a latency problem is not new, it is the technique of choice in many high speed computers. There are many algorithms which will run efficiently on multiprocessors if vector operations can be made to run at full bandwidth. In the standard switch implementations considered to connect

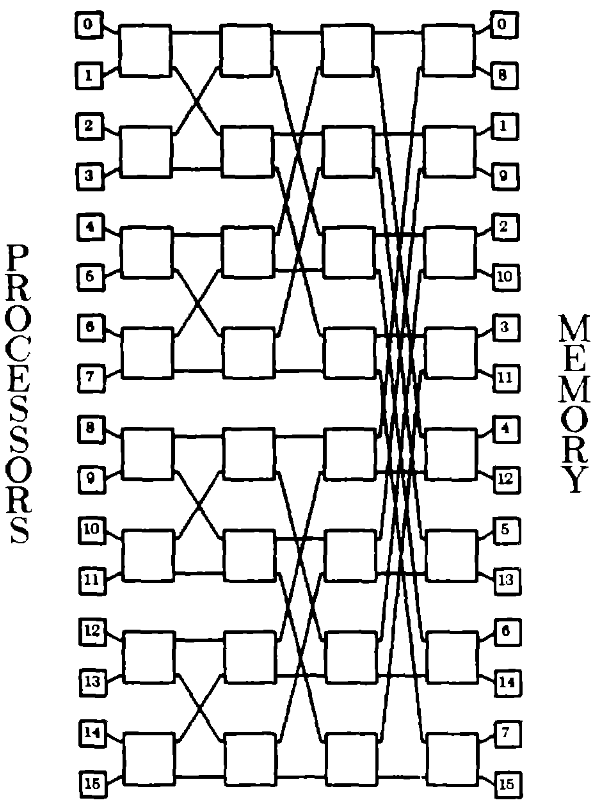

Figure 1

The butterfly multiprocessor architecture.

processors to memory modules for large N multiprocessors, fluctuations in timing caused by conflicts lead to further conflicts in the switch. This chaotic behavior causes a bandwidth reduction for vector operations. By introducing a modification to the standard 2x2 switch node used in such networks, we construct a processor to memory connection which will adaptively remove the chaotic behavior caused by conflicts. With the new switch design, the system will eventually fall into lock step during a simultaneous vector operation by all of the processors. If the vectors are long enough the startup overhead can be amortized and full utilization of the processing power of the machine will be obtained.

The sections of this paper are as follows. In section 2 we consider the constraints placed on a cpu node by the behavior of a packet switched memory server. The fundamental problem that the processor must deal with is the arbitrary latency which a packet might incur as it is routed to and from a memory module. In section 3 we consider the performance of a memory server constructed from the standard 2 buffer switch node appearing in the literature. We simulate the action of the memory server for both vector gather/scatter operations and vector loads/stores with strides. For vector operations with strides, starting at random addresses, this memory server delivers disappointing results which are similar to those for gather/scatter addressing. In section 4 we introduce our modified 2x2 switch node with 4 internal buffers. For random addressing, a memory server made of these nodes delivers a normalized bandwidth approaching unity as the lengths of the buffers are increased. When we consider vector loads and stores, with random starting addresses, we get surprising results. The memory server adaptively removes the chaotic behavior caused by conflicts and after a settling time achieves a normalized bandwidth of unity. Every processor, or memory module for vector stores, receives a vector element on each clock cycle. Finally, in section 5, we discuss these results and their implications for the future of large N common memory multiprocessors.

2. The architecture of the cpu node

In order to put our simulations into context, we consider a cpu architecture that is useful in conjunction with a packet switched memory server. The multiprocessor consists of N identical cpu nodes each of which is connected to the memory server through a port. The operation of the port is pipelined and it is capable of receiving and transmitting a packet on every machine cycle. The multiprocessor as a unit is fully synchronous. All packets are of a fixed size and a single clock is distributed to all components. We envision a RISC architecture [2] for the cpu node, with vector instructions, that cleanly separates memory operations from the other functions of the cpu. As we describe the architecture of the cpu node, which is shown in figure 2, we will point out the features that were introduced in order to obtain good performance when used with a packet switched memory server.

The memory server, viewed from the perspective of the cpu, is a port which packets can be transmitted to and received from. The packet which is transmitted by the cpu, which we will call a *request*, contains information which identifies the cpu making the request, the memory module of which the request is being made and the particular function to perform. The packet switching network routes the request to the appropriate memory module solely on the basis of the information contained within the packet. When the request arrives at its destination, a new packet called a *response* is routed back to the cpu originating the request. If the request was a *read* the response will contain the desired data. If the request was a *write* the response simply notifies the cpu that the operation has been completed. This return receipt for write requests is important when we synchronize various processors and will be discussed again later. One does

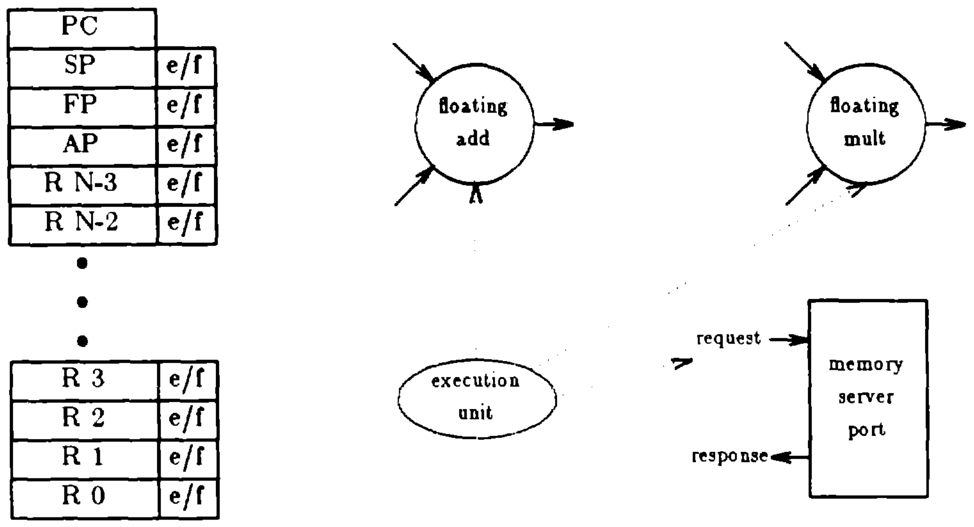

Figure 2

A cpu architecture suitable for multiprocessors.

need to do more than just read and write data. Among these operations are synchronization operations and one allowing the processor to inquire about its position in the network. All of these operations fit into the general request response scheme.

In order to understand the reason for the fundamental feature of the processor, the clean separation of memory operations and data manipulation functions, consider how the cpu fetches operands from the memory server. Suppose we want to load register R0 with a word from memory at address A. In executing this instruction, the execution unit marks the register R0 empty and puts a read request on the port to the memory server. Once the read request packet is dispatched the execution unit is free to begin another instruction. The register R0 will be filled some indeterminate number of clocks later without further attention by the execution unit. Should a later instruction require register R0, and it is still empty, the execution unit will wait until it is filled. Through careful reordering of the separate arithmetic operations and memory requests, one can mask some of the latency that occurs in packet switched memory servers. If contiguous sets of registers are manipulated as vectors, and full bandwidth can be maintained through the switch, the latency of the memory server will be fully masked for long vectors.

How does the register get filled? The request packet contains a token indicating the desired operation, in this case *"load register R0 with a word"*, the address A and the identity of the cpu making the request. The switch uses the address A to route the packet to the memory module containing the desired word. Once the request packet arrives at the memory module, an answer packet is constructed which contains the desired data, the operation token and the identity of the cpu making the request. The cpu identity is used to route the answer back to the cpu originating the request. When the answer reaches the cpu originating the request the data is deposited in register R0 independently of the operation of the execution unit. As the packet may conflict with others sent by different cpu nodes, the delay for a load request can not be determined in advance. Furthermore, the ordering of the answer packets is not guaranteed to be the same as the request packets. Because of this, the request and answer packets must carry the register destination and each register of a group being used as a vector must have its own empty full state.

In order to be viable for use in a multiprocessor our cpu node must also have synchronization instructions. These operations which might include "test and set" and "fetch and add", among others, are implemented by special request packets. Due to the possibility of a synchronization operation beating a memory operation through the switch, return receipt packets are included for write requests. A counter in the cpu node keeps track of the pending read or write requests and a synchronization instruction waits until all memory requests have been satisfied. By using a return receipt mechanism for all memory operations the integrity of shared data can be guaranteed.

3. The performance of the standard switch

Consider a memory server constructed from the 2x2 switch node shown in figure 3. Packets enter the two input ports on the left and feed the internal buffers. A single bit of the packet is used to determine the output port that will be used to exit the switch. If this bit is 0 then the packet will exit output port 0. Otherwise the packet will exit output port 1. The heads of the two buffers compete for access to the output ports. If both packets are destined for the same output port then the packet in the buffer fed by port 0 aways wins the conflict and the other waits while it moves to its destination port. At a cost of extra hardware one can let the age of the packets determine the winner of any conflicts. We have found that this changes the performance of the network very little.

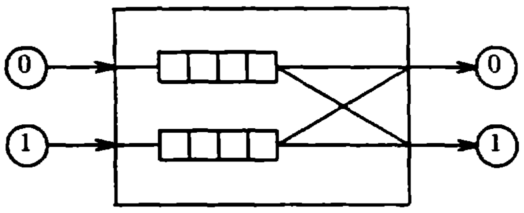

Figure 3

The standard 2 buffer switch node.

Using this switch model we simulated a memory server for a saturated random addressing load, which would occur for a gather/scatter vector operation with random addresses, and for a stride 1 vector load with random starting addresses. The system was started with an empty network and all processors began making requests simultaneously. In table 1 we show the normalized bandwidth, the bandwidth divided by N, as a function of network size and buffer length. The poor performance of the switch for a buffer size of 1 is clearly demonstrated. As the buffer size is lengthened the normalized bandwidth approaches 75%. This would not be a bad performance level for random gather/scatter addressing but we would prefer that the normalized bandwidth approach 100% as the buffer length is increased. When stride 1 vector addressing with random starting addresses was simulated we obtained results similar to those shown in table 1. The conflicts caused by the random starting addresses led to chaotic behavior. Although 75% of ideal performance might be acceptable even for vector operations with strides, we would prefer a multiprocessor system that could adaptively remove the conflicts and eventually achieve ideal bandwidth.

Normalized random fetch bandwidth						
N	buffer length					
	1	2	4	8	16	32
2	.40	.75	.75	.75	.75	.75
4	.31	.59	.67	.71	.73	.74
8	.25	.51	.63	.69	.72	.74
16	.22	.46	.60	.67	.71	.72
32	.20	.43	.58	.67	.70	.72

Table 1

Normalized random fetch bandwidth for the standard 2 buffer switch node.

4. The modified 4 buffer switch node

Why is the bandwidth of the 2 buffer switch node limited to 75%? Examining figure 3 we can see the root of the problem. If the two packets at the heads of the buffers need to be routed to the same output port one of them must wait for the next clock cycle. The waiting packet blocks any packet sitting further back in the buffer which is destined for the unused port. If we could implement a 2x2 switch that could manage to slip a packet past this block, full bandwidth would be maintained. In figure 4 we show a 2x2 switch capable of performing this feat. This switch has 4 internal buffers. Packets entering the input ports are sorted into the buffers according to the output port they are destined for. With this presorting of packets into separate buffers the only way we can have a blocked port is to have zero packets in the switch destined for the port. If the buffers are long enough this is unlikely to happen. As with the previous switch, we tried both a hard wired priority where a buffer fed by port 0 always won any conflict and one where the age of the packet influenced the priority. Although the priority mechanism using packet age was slightly better, the difference in performance was not very large.

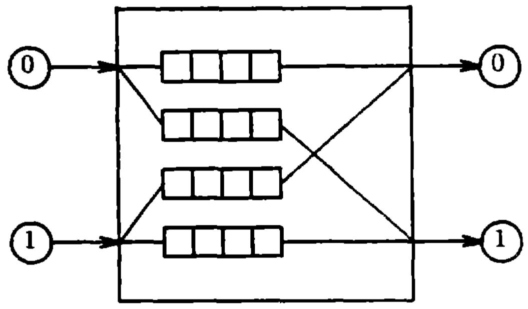

Figure 4

The modified 4 buffer switch.

It is not surprising that a butterfly network constructed from this new switch node gives a limiting normalized bandwidth of 1 for gather/scatter operations with random addressing. In table 2 we show the bandwidth of the system for such operations as a function of N and the buffer length. When we tried stride 1 vector operations, with random starting addresses, we would have been happy with similar results. Instead the new network delivered a surprise. After an initial settling in period the system fell into lock step with every cpu getting a vector element each clock cycle. The new switch adaptively absorbed the conflicts caused by the random starting addresses and eventually reached a bandwidth of 1.

The vector elements, however, do not arrive in the exact order of request. They arrive in a *bunny hop*† which is dependent on initial conditions. The approximate ordering allows chaining of arithmetic operations if the cpu is properly designed. Once the cpu has waited for a vector element that has incurred the maximum latency in the repeating sequence it will never have to wait again for a vector element as it proceeds with the chained operation.

The result is not restricted to a stride of 1. If suitable constraints on the initial starting addresses are met, for instance if the stride is 2 then exactly half of the starting addresses must be even, the network locks in to a normalized bandwidth of 1 for any stride. This restriction on the starting addresses is just the one required for full bandwidth on a crossbar. If the start addresses are unconstrained and the stride is a power of two the network achieves lock in at some bandwidth less than unity. There are unfortunately some pathological addressing cases. A

†A very dated dance craze.

stride of 3, 5, 7 or some other number that does not cleanly factor into the number of nodes in the network combined with unconstrained starting addresses can lead to a real bandwidth disaster.

Normalized random fetch bandwidth						
N	buffer length					
	1	2	4	8	16	32
2	.40	.83	.92	.96	.98	.99
4	.31	.66	.86	.93	.96	.98
8	.25	.60	.82	.91	.94	.96
16	.22	.55	.79	.89	.93	.96
32	.20	.53	.77	.88	.93	.96

Table 2

Normalized random fetch bandwidth for the 4 buffer switch node.

The first question that one asks, given the results discussed above for vector operations, is how long must the vectors be. In table 3 we provide the answer for stride 1 vectors with random starting addresses. The column labeled by $n_{1/2}$, $n_{3/4}$ and $n_{.9}$ give the vector lengths required for an average normalized bandwidth of ½, ¾, and .9 respectively. The column labeled L_b gives the buffer length required to achieve lock step. As can be seen in the table, the vector lengths required for efficient operation do not grow too rapidly with the number of cpu nodes N. The buffer lengths required grow very modestly with N.

Performance for stride 1 vector loads					
$\log_2 N$	N	$n_{1/2}$	$n_{3/4}$	$n_{.9}$	L_b
1	2	2	6	19	1
2	4	7	21	64	2
3	8	11	33	100	2
4	16	14	48	145	2
5	32	21	72	226	4
6	64	34	114	370	6
7	128	50	177	550	12
8	256	84	225	712	12
9	512	71	297	1099	21
10	1024	88	393	1396	20
11	2048	110	453	1783	NA

Table 3

Vector lengths required for stride 1 vector operations.

It is interesting to consider the functional dependence of the vector lengths $n_{1/2}$, $n_{3/4}$ and $n_{.9}$ on N for large N. By plotting n vs $\log_2 N$ on a log-log plot we find that the points fit a power curve very well for large N. Performing a standard power curve fit to the last 8 points we find the following relationships.

$$n_{1/2} = .850(\log_2 N)^{2.04} \tag{1}$$

$$n_{3/4} = 1.91(\log_2 N)^{2.30} \tag{2}$$

$$n_{.9} = 4.09(\log_2 N)^{2.52} \tag{3}$$

The quality of the power curve fit is clearly indicated by the plot of $n_{3/4}$ vs $\log_2 N$ shown in figure 5. We are well into the large N regime with our simulations.

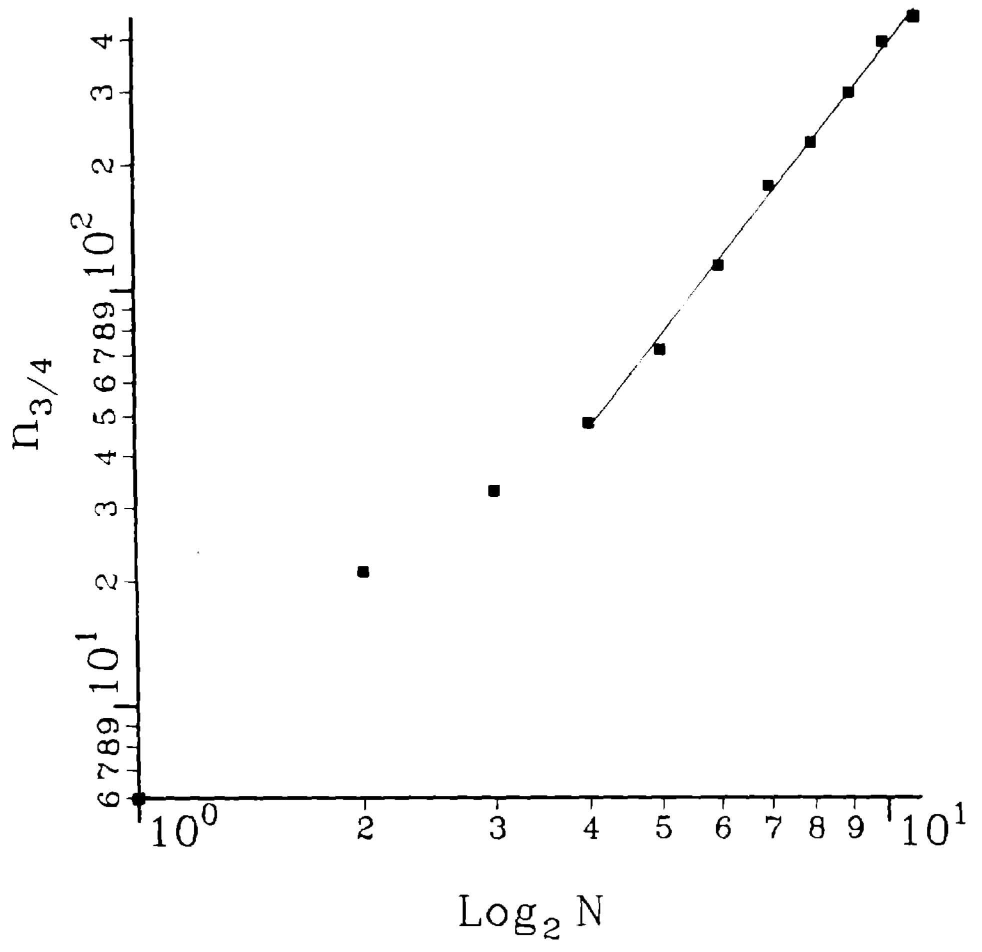

Figure 5

Plot of $n_{\frac{3}{4}}$ vs $\log_2 N$.

5. Conclusions

We have presented a modified butterfly memory server network that is capable of delivering full bandwidth to a large number of processors performing simultaneous vector operations. The memory server adaptively absorbs conflicts and timing fluctuations making it suitable for general purpose large N pipelined multiprocessors. Scalar fetches through the memory server still incur a latency of $\alpha \log_2 N$ which limits the scalar performance of the machine. The switch saturation caused by vector operations will reduce scalar fetch performance even further. In a real machine one would of course include local memory in order to increase speed for scalar data which does not need to be shared. It may also be profitable to include a separate memory server switch for scalar fetches from shared memory. Examining these issues is beyond the scope of this paper.

We have not investigated Banyan networks [5] to see if the same basic switch node modification yields adaptive behavior for vector operations. This is an interesting proposition as these networks offer a much lower latency of $\alpha \log_j N$ where j is the number of ports on the basic switch node. The lower memory latency would improve the speed of scalar operations and would reduce the vector lengths required for efficient vector performance. We will report on Banyan and other networks with interesting features in a future paper.

It is very encouraging that pipelined arithmetic can be efficiently used to mask memory latency in large N common memory multiprocessors. The constraint placed on the code for such machines, the use of vectorization, is the same one that users of high speed computers deal with now. We have presented a shared memory architecture which can give full memory bandwidth for vectorized problems. Will computer manufacturers build us one?

References

[1] T. S. Axelrod, "Effects of Synchronization Barriers on Multiprocessor Performance," submitted to: IEEE Trans. Comput.

[2] D. A. Patterson, "Reduced Instruction Set Computers," Commun. ACM, Vol. 28, Num. 1, pp. 8-21 (Jan. 1985).

[3] J. H. Patel, "Performance of Processor-Memory Interconnections for Multiprocessors," IEEE Trans. Comput., vol. c-30, pp. 771-780 (Oct. 1981).

[4] D. M. Dias and J. R. Jump, "Analysis and simulation of buffered delta networks," IEEE Trans. Comput., vol. c-30, pp.273-282 (Apr. 1981)

[5] S. Cheemalavagu and M. Malek, "Analysis and Simulation of Banyan Interconnection networks with 2x2, 4x4 and 8x8 Switch Elements," Proc. Real-Time Systems Symposium, pp.83-89, Los Angeles, California (Dec. 1982).